Automatic mangling rules generation

Simon Marechal (bartavelle at openwall.com)
http://www.openwall.com
@Openwall

December 2012

Mangling rules ?

- First seen in Crack
- Transform (or reject) a dictionary word
- Almost all major tools use a language derived from Crack's
  - Look at hashkill's language for a fancy exception !

A John the Ripper default ruleset

```
<* >2 !?A l $[2!37954860.?]
```

This ruleset is processed into 12 rules, one for each character between the brackets, such as:

```
<* >2 !?A l $2
```

This will reject the word unless it is shorter than the maximum length, is of at least two characters long, and is purely alphabetic. If the word is not rejected, this rule will convert the word to all-lowercase, and append the character "2".

John the Ripper mangling rule language

John the Ripper has one of the most complete languages:

- Preprocessor
- All kinds of substitutions, insertions, deletions
- Duplicate, reflect, pluralize ...
- Memory access commands (recently also in hashcat)
- Word rejection rules, as well as rule rejection flags
- And many others

It can emulate most of other tools specific rules using multiple rules

Converting rules between engines

The *manglingrules* library – `http://openwall.info/wiki/john/rulesfinder`

- Only converts between hashcat and John the Ripper
- Does not convert bitwise shifts, ASCII increments, or 'q'
  - Possible, but clunky
- But should convert everything else

The problem

How to automatically generate a *good* set of mangling rules ?

- Machine learning problem
- Given a training set and wordlist, generate the mangling rules
- *Good* means that it cracks the passwords in the test sets

Requirements

- Training set (list of cleartext passwords)
  - Full rockyou (32.6M entries)
  - Rockyou without dupes (14.3M entries)
- Training wordlist
  - Wikipedia-sraveau (58.4M entries)
  - Lowercase dictionary (same as above, filtered, 24.4M entries)
- Test set: the 2012 Yahoo Contributor Network leak
  - 453491 distinct passwords
  - 342514 unique passwords
  - Unique passwords used, to reduce biases (and introduce new ones, hopefully less problematic)

Naïve algorithm

1. For each element of the training set
   - Find the closest word in the wordlist
   - Compute the shortest mutation rule that goes from one to the other
2. Compute how many passwords each rule cracks
3. Sort them by effectiveness
4. ???
5. Profit !

The only problem is that these steps are either hard to define or too computationally expensive ...

Hard problems

- How to find the closest word ?
- How to find the shortest mutation rule ?
- Why just the closest word ?
- Why just one rule ?
- What if several rules crack the same password ?
- How to sort the rules ?

- Someone chooses a base word,
- mangles it,
- then appends and prepends optional stuff.

Real world example

- Choose a base word → `linkedin`
- Leetify → `l1nk3d1n`
- Append/prepend stuff → `g0l1nk3d1n!`

Which weakness of this model is demonstrated here ? The user certainly added "go" before leetifying.

Selected algorithm

- Start with a list of mangling rules
  - Hand picked, computed or randomly generated
- Mangle the wordlist
- For each element of the training set
  - Find the largest substring that matches the mangled words
  - Compute appends/prepends
- Only keep the best full rules (base rule + appends/prepends)
- Rank the rules by number of passwords cracked (not yet by efficiency)

Picking rules

2789 reasonably useful rules right now

- From known sets
- Exhaustive application of single rules (overstrike, insert, etc.)
- A few combinations of base rules
- Several underperforming rules that need to be pruned
- Obviously no append/prepend rule

Note that it is a good (and easy) idea to steal rules from others !

Mangling / searching for the substring

The wordlist is mangled with JtR, and is fed into the search program, which:

- Loads the mangled data
  - Computes CityHash[1] and a rolling hash[2] for each word
  - Populates two Bloom filters with the two results
  - Stores each entry in an AVL tree (sorting by CityHash)
- For each cleartext, finds the largest substring matching the mangled data
  - No good algorithm that does exactly this
  - Using N rolling hashes, for matches at most N characters long
- Finds and stores prefixes and suffixes

Optimized for efficiency: no disk access after initial loading, resides in available memory

[1] `http://code.google.com/p/cityhash/`

[2] Actually not a "real" rolling hash, but serves the same purpose

Compiling rules

Example:

- Rule `T0`, cleartext "`Pass123`"
- The dictionary word "`pass`" is mangled into "`Pass`"
- It is the longest match in our cleartext "`Pass`123"
- Thus, the rule `T0 Az"123"` cracks "`Pass123`"

All these rules are collected for all plaintexts, and (rule, cleartext[3]) pairs are stored on disk. Then a second pass uses this to build a list of pairs (unique rule, list of cleartexts).

[3] Actually its index

Ranking the rules

Start with a list of pairs (unique rule, list of cleartexts), then

1. Sort it by the size of the cleartexts list
2. Take the top rule
3. Remove the cleartexts cracked by this rule from the others
4. Rinse and repeat

This is a greedy approximation of the optimal solution of the maximum coverage problem

Other approaches – best64 challenge

Hashcat competition using the phpbb.com leak as training AND validation set, and "top 10k" as the wordlist

- Had all my toolset ready at the time, but was a wake-up call (only would have achieved 7th place)
- Found tons of omissions, bugs in rule list
- Arex1337 won using a method pretty similar to what is described here
  - without append/prepend detection
  - a lot of automatic rule generation
  - probably poor runtime performance (PHP + MySQL), but sufficient for the task
  - Many ugly rules in the winning set : `$2$4]]`
  - Ok, maybe only the last step was similar ...

Cracking the July 2012 Yahoo leak, using rockyou and wikipedia-sraveau as wordlists

Rules trained as described previously

Identifier	Training set	Training wordlist
Rockyou	Rockyou	wikipedia-sraveau
Rockyou unique	Rockyou (unique)	wikipedia-sraveau
Lower dict.	Rockyou (unique)	wikipedia-sraveau, lowercased and cleaned

And also, appearing as dotted lines in the figures:

- JtR "jumbo" rules
- Hashcat best64
- Hashcat "generated"
- Hashcat "d3ad0ne"
- Arex1337 best80

Using rockyou as the wordlist

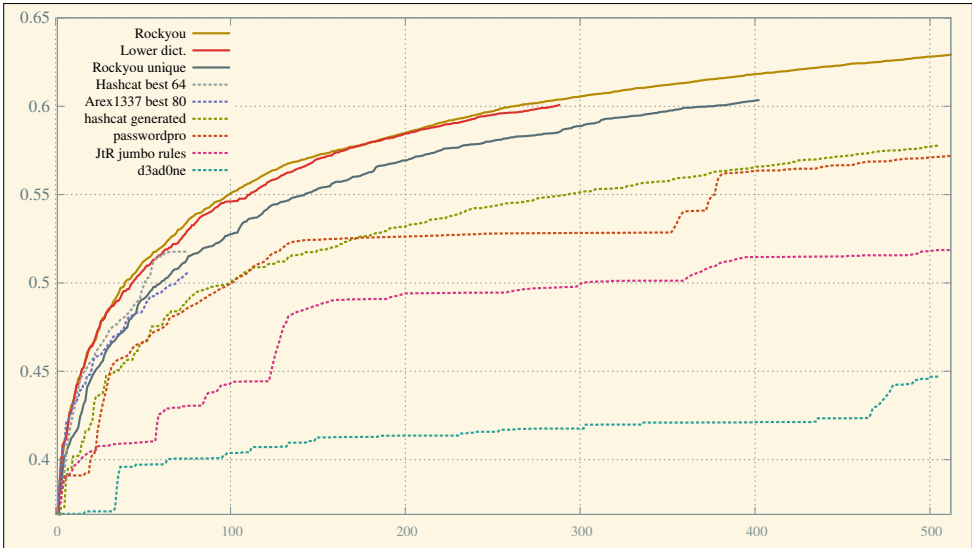

Figure: Passwords cracked per mangling rules processed

Using rockyou as the wordlist – 64 rules

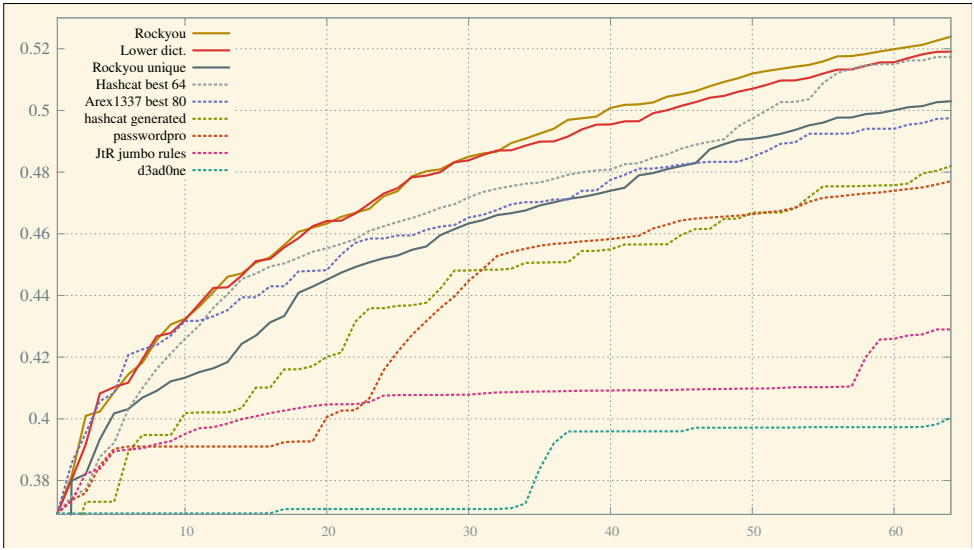

Figure: Passwords cracked per mangling rules processed

Using wikipedia as the wordlist

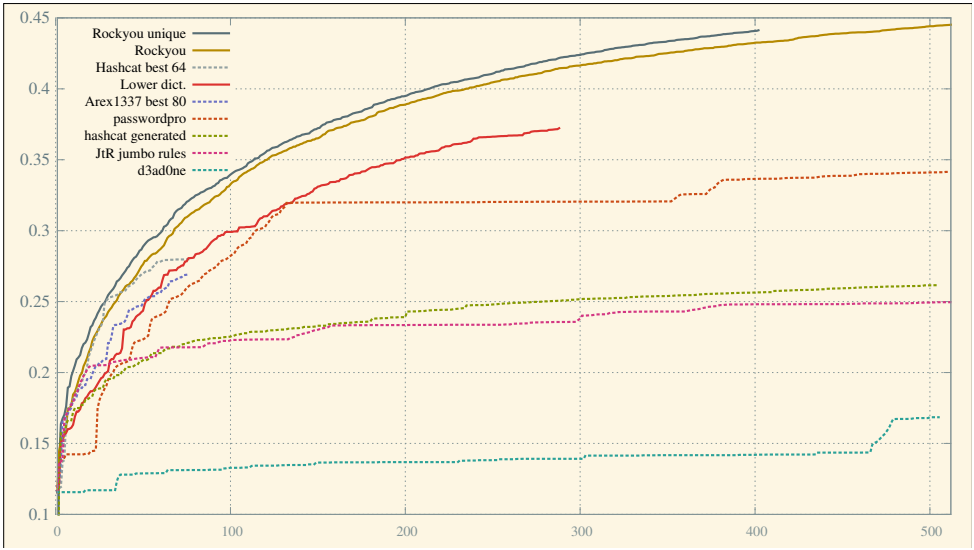

Figure: Passwords cracked per mangling rules processed

Using wikipedia as the wordlist – 64 rules

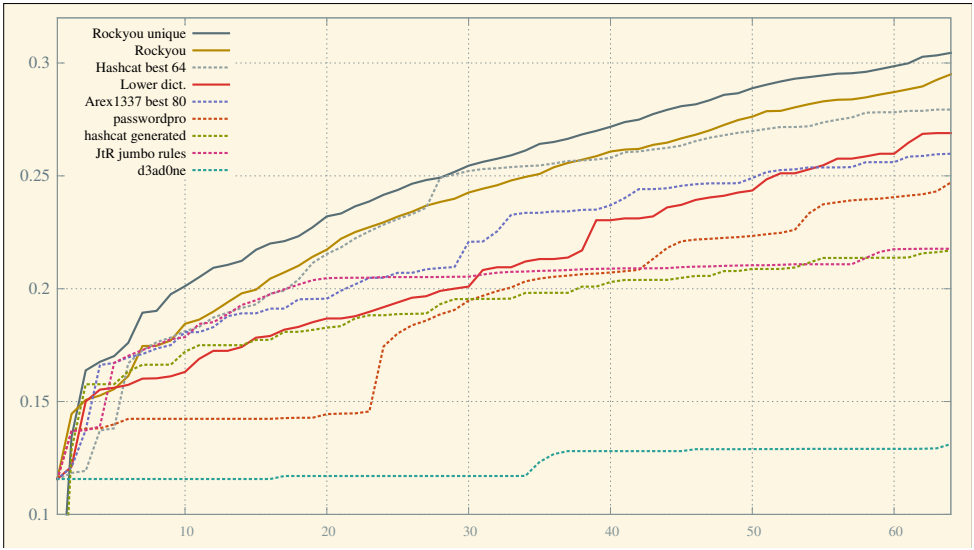

Figure: Passwords cracked per mangling rules processed

Conclusions

- The toolset is satisfying
  - Project page: `http://openwall.info/wiki/john/rulesfinder`
  - Includes the (undocumented and ugly) benchmarking scripts used to create the figures
  - Look out for memory usage !
- Could still be optimized
  - Runtime optimization : pre-apply known best rules
  - Performance optimization : add more effective base rules, sort rules by tests/cracks ratio
- Against this kind of passwords
  - JtR default rules need an overhaul[4]
  - Hashcat best64 is good, but only has 64 rules

[4] Please note that rejection rules were not taken into account in the figures

Bonus – good rules are strange to look at

1. `:`
2. `$1`
3. `]`
4. `S`
5. `D3`
6. `$2`
7. `Az"123"`
8. `c`
9. `D5`
10. `Az"12"`
11. `D2`
12. `$3`
13. `$7`
14. `Az"13"`
15. `D4`
16. `Az"11"`
17. `$5`
18. `[`
19. `Az"22"`
20. `Az"23"`
21. `Az"01"`
22. `Az"21"`
23. `Az"07"`
24. `$4`
25. `A0"1"`
26. `} } [ [`
27. `Az"10"`
28. `Az"14"`
29. `Xm1z`
30. `Az"08"`
31. `Az"06"`
32. `Az"15"`
33. `$8`
34. `[ A0"j"`
35. `r`
36. `Az"69"`
37. `$6`
38. `Az"16"`
39. `Az"18"`
40. `$9`
41. `$!`
42. `S $1`
43. `Az"17"`
44. `} } } }`
45. `Az"24"`
46. `Az"05"`
47. `[ A0"m"`
48. `Az"09"`
49. `Az"88"`
50. `l p`
51. `] d`
52. `c $1`

Final steps

Before being usable, the rules list must be processed. Each line looks like :

```
:  Az"1" NBPWD=169049
```

- The `NBPWD=169049` indicates how many passwords this rule cracked in the training set, but must be removed before the rule is used by John the Ripper
- Some rules could be optimized, for example `[ A0"m"` could be written `o0m`
- In order to be usable by another tool, some rules will need to be converted

The `manglingrules` library previously discussed can help with that, but is still experimental. It can:

- Parse most hashcat and JtR rules (it lacks some JtR preprocessor directives, and the most recent hashcat rules) into its internal representation
- Optimize this internal representation based on a simple pattern system
- Convert the rules to the JtR or hashcat flavor

Questions?

http://www.openwall.com
bartavelle at openwall.com